International Computer Science Competition

Qualification Round Solutions

Sumaiya Rahman Soma

August 2025

Problem A : Neural Network Component Identification

Symbolic Representation

- $w_{21}^{(1)}$: A **weight**, representing the strength of the connection between neurons.
- Σ: The **summation node**, which computes the weighted sum of inputs plus bias.
$$z_j^{(l)} = \sum_i w_{ij}^{(l)} a_i^{(l-1)} + b_j^{(l)}$$
- f: The **activation function**, applied element-wise to introduce non-linearity:
$$a_j^{(l)} = f\left(z_j^{(l)}\right)$$
- Red circle (•): Represents the **input layer** neurons (e.g. Duration stayed, Tip received).
- Orange circle (•): Represents the **bias unit**, which shifts the activation function.
- Green circle (•): Represents the **output layer** neuron, producing the prediction $\hat{y}$.

Layer Identification

- Box A: **Hidden layer**, where intermediate computations occur after applying weights, summation, and activation.
- Box B: **Output layer**, producing the final prediction $\hat{y}$.
- $\hat{y}$: The **predicted output** of the network.

1. Inputs x_1, x_2 (Duration stayed, Tip received) enter the network.
2. Weighted sums are computed:
$$z_j^{(1)} = \sum_i w_{ij}^{(1)} x_i + b_j^{(1)}$$

3. Hidden activations are calculated:

$$a_j^{(1)} = f\left(z_j^{(1)}\right)$$

4. These activations are passed to the output neuron:

$$z^{(2)} = \sum_j w_j^{(2)} a_j^{(1)} + b^{(2)}$$

5. Finally, the output is produced:

$$\hat{y} = f\left(z^{(2)}\right)$$

Problem B: Cake Calculator

Pseudocode

Below is the implementation of Problem B:

```
Function cake_calculator(flour, sugar):
    FLOUR_PER_CAKE = 100
    SUGAR_PER_CAKE = 50

    # Step 1: Compute maximum cakes based on
       available ingredients
    max_cakes = min(flour / FLOUR_PER_CAKE, sugar /
       SUGAR_PER_CAKE)

    # Step 2: Compute leftover ingredients
    remaining_flour = flour - (max_cakes *
       FLOUR_PER_CAKE)
    remaining_sugar = sugar - (max_cakes *
       SUGAR_PER_CAKE)

    # Step 3: Return results as a list
    return [max_cakes, remaining_flour,
       remaining_sugar]
```

Listing 1: Cake Calculator Pseudocode

Explanation

The algorithm calculates the maximum number of cakes that can be baked from a given amount of flour and sugar and computes the leftover ingredients. Each cake requires 100 units of flour and 50 units of sugar. The inputs are assumed to be non-negative integers.

Maximum Cakes (Integer Division)

- Compute cakes possible from flour alone: flour/100.
- Compute cakes possible from sugar alone: sugar/50.

- Take the minimum of these two values to ensure neither ingredient runs out:
$$\text{max\_cakes} = \min\left(\lfloor\frac{\text{flour}}{100}\rfloor, \lfloor\frac{\text{sugar}}{50}\rfloor\right)$$
- Integer division automatically floors the result, giving the maximum number of complete cakes.

Leftovers After Baking

- Let max_cakes be the number of cakes baked.
- After baking max_cakes :

  $\text{remaining\_flour} = \text{flour} - (\text{max\_cakes} \cdot 100), \quad \text{remaining\_sugar} = \text{sugar} - (\text{max\_cakes}$
- This ensures leftovers are non-negative and valid.

Edge Cases

- If flour < 100 or sugar < 50, then max_cakes $= 0$, and the remaining ingredients are the input amounts.
- The algorithm handles large inputs efficiently since it performs constant-time calculations.

Why This Solution is Better Than a Loop

- **Constant-time solution ($O(1)$)**: Using $\min(\text{flour}/100, \text{sugar}/50)$ computes the maximum cakes in a single step. A loop would take $O(\text{cakes})$ steps.
- **Correctness guaranteed**: The integer division floors the number of cakes, and the subtraction formula ensures accurate leftovers.
- **No need for dynamic programming**: Each cake consumes a fixed amount, so there is no overlap in subproblems and only one optimal solution (the minimum of the two constraints).
- **Efficient and scalable**: Works for any nonnegative input, producing the maximum number of cakes and exact leftovers in constant time.

Problem C : The School Messaging App

Solution for Question 1:

Standard text encodings use the same number of bits for every character, which is simple but often **inefficient** when some characters appear more frequently than others. Fixed-length codes must allocate enough bits to uniquely represent all characters:

$$\text{bits needed} = \lceil \log_2(\text{number of characters}) \rceil$$

- **Example:** 4 characters (A, B, C, D) $\rightarrow \lceil \log_2 4 \rceil = 2$ bits per character.

However, using **variable length codes** allows us to assign:

- Shorter codes to frequent characters
- Longer codes to rare characters

This reduces the **average number of bits per character**, making communication more efficient and allowing more messages to be transmitted under limited bandwidth.

Example

Character	Probability	Variable-Length Code	Length (bits)
A	0.5	0	1
B	0.25	10	2
C	0.15	110	3
D	0.10	111	3

The average code length is calculated as:

$$L = 0.5 \cdot 1 + 0.25 \cdot 2 + 0.15 \cdot 3 + 0.10 \cdot 3 = 1.85 \text{ bits/character}$$

Notice that the average length 1.85 bits is less than the 2 bits required by fixed length codes.

Reasoning

- Frequent symbols dominate the total length of the message, so assigning them shorter codes **saves more bits, in general**.
- Rare symbols occur less frequently, so their longer codes have minimal impact on the average.

By tailoring the lengths of the codes to the probabilities of character, the variable length codes **maximize data efficiency**, reduce the average size of the message, and allow more messages to be sent under strict bandwidth constraints.

Solution for Question 2: Entropy of the 12-character set

Given the symbol probabilities,

$$\Pr(A) = 0.20,\ \Pr(B) = 0.15,\ \Pr(C) = 0.12,\ \Pr(D) = 0.10,\ \Pr(E) = 0.08,\ \Pr(F) = 0.06,$$
$$\Pr(G) = 0.05,\ \Pr(H) = 0.05,\ \Pr(I) = 0.04,\ \Pr(J) = 0.03,\ \Pr(K) = 0.02,\ \Pr(L) = 0.10,$$

the (base-2) entropy is

$$H(X) \ = \ -\sum_{i=1}^{12} p_i \log_2 p_i \ = \ \sum_{i=1}^{12} p_i \log_2\Big(\tfrac{1}{p_i}\Big)\,.$$

Compute each term $p_i \log_2(1/p_i)$ (rounded to 6 decimal places):

$$\begin{aligned}
0.20\log_2(5) &= 0.464386,\\
0.15\log_2(6.\overline{6}) &= 0.410545,\\
0.12\log_2(8.\overline{3}) &= 0.367067,\\
0.10\log_2(10) &= 0.332193,\\
0.08\log_2(12.5) &= 0.291509,\\
0.06\log_2(16.\overline{6}) &= 0.243534,\\
0.05\log_2(20) &= 0.216096,\\
0.05\log_2(20) &= 0.216096,\\
0.04\log_2(25) &= 0.185754,\\
0.03\log_2(33.\overline{3}) &= 0.151767,\\
0.02\log_2(50) &= 0.112877,\\
0.10\log_2(10) &= 0.332193.
\end{aligned}$$

Summing the contributions gives the entropy:

$$\begin{aligned}
H(X) &\approx 0.464386 + 0.410545 + 0.367067 + 0.332193 + 0.291509 + 0.243534\\
&\quad + 0.216096 + 0.216096 + 0.185754 + 0.151767 + 0.112877 + 0.332193\\
&\approx \boxed{3.324016 \text{ bits/character}}.
\end{aligned}$$

Interpretation (optimal encoding)

The entropy $H(X) \approx 3.324016$ bits/character is the *theoretical lower bound* on the expected (average) number of bits per symbol for any lossless encoding of this source. Concretely:

- No encoding can have an expected length smaller than $H(X)$.
- For any prefix (instantaneous) code with average length $\bar{L}$,

$$H(X) \;\leq\; \bar{L} \;<\; H(X) + 1.$$

- Practical optimal codes (e.g., Huffman) produce $\bar{L}$ close to $H(X)$; the difference $\bar{L} - H(X)$ is the coding overhead.
- Compared to fixed-length coding that requires $\lceil \log_2 12 \rceil = 4$ bits / symbol, an entropy-guided variable-length code can approach $\approx$ 3.324 bits/symbol on average, saving bandwidth.

Thus, $H(X)$ quantifies the best possible average compression for the given probability distribution.

Solution for Question 3: Average length of the Fano code and comparison to entropy

Given Fano codes and lengths

Symbol	Fano code	ℓ_i (length in bits)
A	000	3
B	100	3
C	010	3
D	1100	4
E	0110	4
F	1010	4
G	001	3
H	1011	4
I	0111	4
J	1101	4
K	1111	4
L	1110	4

Average code length

The average code length for the prefix code is

$$\bar{L} = \sum_{i=1}^{12} p_i \, \ell_i.$$

Using the given probabilities and lengths:

$$\begin{aligned}
\bar{L} &= 0.20 \cdot 3 + 0.15 \cdot 3 + 0.12 \cdot 3 + 0.10 \cdot 4 + 0.08 \cdot 4 + 0.06 \cdot 4 \\
&\quad + 0.05 \cdot 3 + 0.05 \cdot 4 + 0.04 \cdot 4 + 0.03 \cdot 4 + 0.02 \cdot 4 + 0.10 \cdot 4 \\
&= 0.60 + 0.45 + 0.36 + 0.40 + 0.32 + 0.24 + 0.15 + 0.20 + 0.16 + 0.12 + 0.08 + 0.40 \\
&= \boxed{3.48 \text{ bits/character}}.
\end{aligned}$$

Comparison with entropy

From Question 2, the (base-2) entropy of the source is

$$H(X) \approx \boxed{3.324016 \text{ bits/character}}.$$

Gap and efficiency

$$\text{Redundancy (gap)} = \bar{L} - H \approx 3.48 - 3.324016 = 0.155984 \text{ bits/char.}$$

$$\text{Coding efficiency} = \frac{H}{\bar{L}} \approx \frac{3.324016}{3.48} \approx 0.9552 \quad (\approx 95.52\%).$$

Interpretation and conclusion

- The Fano code has average length $\bar{L} = 3.48$ bits/character, which is only about 0.156 bits above the theoretical lower bound $H(X) \approx 3.324$ bits/character.
- The code is therefore quite efficient: it achieves roughly 95.5% of the entropy limit.
- Compared to a fixed-length code (which would require $\lceil \log_2 12 \rceil = 4$ bits/character), the Fano code saves $4 - 3.48 = 0.52$ bits/character on average.
- The remaining overhead $\bar{L} - H$ is the typical penalty for using an integer-length prefix code; Huffman codes are optimal among prefix codes and would give a similar average length (within the usual < 1 bit bound).

Solution for Problem D: Word Search Puzzle

- The algorithm is deterministic for word length sorting but randomized for placement positions.
- Maximizing word count favors shorter words after longer ones are placed.
- Overlap allows more efficient usage of grid space.
- Words longer than the grid are ignored to maintain correctness.

Pseudocode

```
Input: words - list of strings
Output: 10x10 grid containing words

Function create_crossword(words):
    size = 10
    grid = 10x10 array filled with '.'

    # Preprocess words: convert to uppercase
    for each word in words:
        word = uppercase(word)

    # Sort words by descending length
    words_sorted = sort(words) by length descending

    # Place each word in the grid
    for each word in words_sorted:
        if length(word) > size:
            continue  # cannot place this word

        placed = False
        for attempt in 0..max_attempts:
            x = random row
            y = random col
            directions = shuffle(all 8 directions)
            for each (dx, dy) in directions:
                if word fits at (x,y) in (dx,dy):
                    place word in grid
```

```
                    placed = True
                    break
            if placed:
                break

    # Fill remaining empty cells with random letters
    for each cell in grid:
        if cell == '.':
            cell = random letter 'A'-'Z'

    return grid
```

Listing 2: Word Search Puzzle Generation

Reasoning

- **Maximizing word count**: Sorting words by descending length ensures longer words are placed first. Short words can then fill remaining gaps or overlap, increasing the total number of words in the grid.
- **Randomized placement**: Trying random starting positions and shuffled directions avoids deterministic patterns and increases successful placement probability.
- **Overlap concept**: Words can share letters if the letter matches the corresponding position in the grid. This allows multiple words to coexist in the same space.
- **Skipping long words**: Words longer than the grid (more than 10 characters) cannot fit, so they are skipped automatically. This prevents failure or invalid placement.
- **Grid filling**: After placing all possible words, empty cells are filled with random letters to complete the 10×10 puzzle.

Edge Cases

- **Empty input list**: The grid will be filled entirely with random letters.
- **All words longer than 10 letters**: No words can be placed; grid is fully random.

- **Words exactly length 10**: Can only be placed horizontally, vertically, or diagonally within bounds; must occupy full row/column/diagonal.
- **Duplicate words**: Each occurrence is attempted independently; duplicates may overlap if letters match.
- **Highly overlapping words**: Overlaps are allowed if letters match; placement may fail if random attempts do not align, but multiple attempts mitigate this.

Time Complexity

- Let W = number of words, L = maximum word length, $N = 10$ (grid size).
- Worst-case: For each word, try every cell (N^2) and 8 directions, checking L cells: $O(W \cdot N^2 \cdot 8 \cdot L)$.
- With constant grid size ($N = 10$), effective complexity is $O(W \cdot L)$ in practice.

Problem Solution for E

Prove that the NAND gate is **functionally complete**, i.e., as any Boolean function can be constructed using only the NAND gate.

Solution and Reasoning

NAND definition

The NAND gate is defined as:

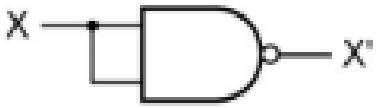
$$\text{NAND}(a, b) = \overline{a \cdot b} = \neg(a \wedge b)$$

where $a, b \in \{0, 1\}$. Its truth table is:

a	b	a NAND b
0	0	1
0	1	1
1	0	1
1	1	0

Express NOT using NAND

A NOT operation can be constructed by connecting both inputs of a NAND gate to the same variable:



$$\neg x = x \text{ NAND } x$$

Check truth table:

x	x NAND x
0	1
1	0

Thus, NAND can implement NOT.

Express AND using NAND

Apply NAND to itself

Let

$$x = (a \text{ NAND } b).$$

Then

$$x \text{ NAND } x = \neg(x \wedge x).$$

Since

$$x \wedge x = x \quad \text{(idempotent law)},$$

we get

$$x \text{ NAND } x = \neg(x).$$

Substitute back

$$(a \text{ NAND } b) \text{ NAND } (a \text{ NAND } b) = \neg(a \text{ NAND } b).$$

But

$$\neg(a \text{ NAND } b) = \neg(\neg(a \wedge b)) = a \wedge b.$$

Thus proved:

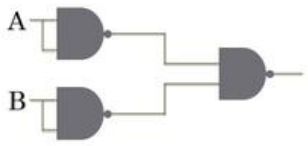
$$a \wedge b = (a \text{ NAND } b) \text{ NAND } (a \text{ NAND } b).$$

Verification using Truth Table

a	b	$a \wedge b$	a NAND b	$(a$ NAND $b)$ NAND $(a$ NAND $b)$
0	0	0	1	0
0	1	0	1	0
1	0	0	1	0
1	1	1	0	1

The final column matches $a \wedge b$, hence the construction is correct.

Express OR using NAND (via De Morgan's Law)



Using De Morgan's law:

$$a \vee b = \neg(\neg a \wedge \neg b)$$

Substitute NOT and AND expressions with NANDs:

$$\neg a = a \text{ NAND } a, \quad \neg b = b \text{ NAND } b$$

$$\neg a \wedge \neg b = (\neg a \text{ NAND } \neg b) \text{ NAND } (\neg a \text{ NAND } \neg b)$$

$$a \vee b = ((a \text{ NAND } a) \text{ NAND } (b \text{ NAND } b))$$

Thus, OR can also be implemented using only NAND gates.

Functional completeness

Since **NOT, AND, OR** are sufficient to construct any Boolean function, and all three can be expressed using only NAND, it follows that:

NAND gate is functionally complete.

Summary of NAND constructions

- **NOT:** $\neg a = a \text{ NAND } a$
- **AND:** $a \wedge b = (a \text{ NAND } b) \text{ NAND } (a \text{ NAND } b)$
- **OR:** $a \vee b = (a \text{ NAND } a) \text{ NAND } (b \text{ NAND } b)$

Conclusion

By constructing NOT, AND, and OR using only NAND gates, we prove that the NAND gate is **functionally complete**. Any Boolean function can therefore be implemented using only NAND gates.